Christian Schuering 1932884

Assignment 2 COSC 3320

(a) To derive a lower bound for the problem of determining whether any element of one list is an element of the other, we can use a comparison-based model of computation. In this model, we assume that the only way to determine whether two elements are equal is by comparing them directly.

Let's consider the worst-case scenario where no elements in the two lists are the same. In order to determine whether any element of one list is an element of the other, we would need to compare each element from the first list with each element from the second list.

For the first element of the first list, we would need n comparisons. For the second element of the first list, we would need n-1 comparisons (since we have already compared the first element). Similarly, for the third element of the first list, we would need n-2 comparisons, and so on.

The total number of comparisons required would be:

n + (n-1) + (n-2) + ... + 2 + 1

This sum is equal to n(n+1)/2, which is a quadratic function in terms of n. Therefore, we can conclude that the lower bound for this problem is Ω(n^2).

(b) To design an algorithm for this problem, we can use a hashing approach. The idea is to insert all the elements of one list into a hash table, and then check if any element from the second list exists in the hash table. This approach has an expected time complexity of O(n).

Here's the algorithm:

1. Create an empty hash table.

2. Insert all elements from the first list into the hash table.

3. For each element in the second list, check if it exists in the hash table.

- If it exists, return true (an element is found in both lists).

4. If no common element is found, return false.

The time complexity of this algorithm depends on the hash table implementation. On average, hash table operations like insertion and lookup take constant time, O(1). Therefore, the overall time complexity of this algorithm is O(n), which matches the lower bound we derived in part (a).

Question 2

Data Structure:

Use an augmented self-balancing binary search tree, such as an AVL tree, similar to the previous approach. The tree nodes will contain the following fields:

value: The value stored in the node.

left: Pointer to the left child node.

right: Pointer to the right child node.

size: The size of the subtree rooted at this node.

Insert(x):

Perform a standard AVL tree or red-black tree insertion to add the element x to the tree.

During the insertion process, update the size field of each node along the insertion path.

Delete(x):

Perform a standard AVL tree deletion to remove the element x from the tree.

During the deletion process, update the size field of each node along the deletion path.

Find([n/5]):

Start at the root node of the tree.

While traversing the tree, compare the size of the left subtree (node->left->size) with [n/5].

If [n/5] is less than the size of the left subtree, move to the left child node and repeat the process.

If [n/5] is greater than the size of the left subtree plus one (accounting for the root node), subtract the size of the left subtree plus one from [n/5] and move to the right child node. Repeat the process.

If [n/5] is equal to the size of the left subtree plus one, we have found the desired node. Return the value stored in that node.

Time and Space Complexities:

The Insert(x) and Delete(x) operations have a time complexity of O(log n) and a space complexity of O(log n) due to the use of an AVL tree or red-black tree.

The Find([n/5]) operation has a time complexity of O(log n) as well since it performs a binary search in the tree. The space complexity is O(1) as it does not require any additional memory beyond the tree structure.

Question 3

function calculateAverageWork(matrices):

n = length(matrices)

S = createMatrix(n, n) // Create an n by n matrix filled with zeros

kCounter = 0

for i = 1 to n - 1:

for j = i to n:

for k = i to j:

S[i][j] += S[i][k] + S[k + 1][j] + rows(matrices[i - 1]) \* cols(matrices[k]) \* cols(matrices[j])

kCounter++

averageWork = S[1][n] / kCounter

return averageWork

// Helper function to get the number of rows in a matrix

function rows(matrix):

return length(matrix)

// Helper function to get the number of columns in a matrix

function cols(matrix):

return length(matrix[1])

// Example usage:

matrices = [ [[1, 2], [3, 4]],

[[5, 6], [7, 8]],

[[9, 10], [11, 12]]

]

averageWork = calculateAverageWork(matrices)

print("Average Work:", averageWork)